EXHIBIT E

Part 2

If we chose the red threshold, then anything more than four login attempts would be considered an anomaly. If we used the blue threshold, then anything more than 5 login attempts would be an anomaly. The green threshold would declare anything over 10 attempts as an anomaly.

Clearly, there is a tradeoff with these thresholds. If we make a low threshold (such as the red one), then on some occasions normal users making innocent mistakes will be viewed as a suspicious anomaly. In particular, if we look at the total odds of a login involving more than 3 attempts in the example above, which we obtain by adding up the size of all columns from 4 out to the right, we find those odds are 3.5%. So in a busy enterprise application with 1000 sessions per day, we would have 35 false alarms per day – enough that busy IT staff would certainly learn to ignore those particular alarms. The proportion of innocent events that will be mistakenly considered to be a problem is known as the *false positive rate*. If we set the threshold to more than 3 as above, we will have a false positive rate of 3.5%. Generally practical systems need to have very low false positive rates, given that the underlying number of events is very large. More than occasional false positives will tend to increase the risk that the system will be ignored. As a commercial practical matter, it is quite hard to achieve satisfactory false positive rates, and the entire industry has struggled with the problem.

On the other hand, if we set the threshold very high, we give would-be attackers more scope than we might wish to guess the password without being detected. For good passwords, brute force will require enormous numbers of guesses that will be easily detectable. However, a certain fraction of users will choose weak guessable passwords given any opportunity to do so, so there is value in minimizing the number of free tries an attacker gets. Situations where the attacker can successfully attack without being detected are known as *false negatives*, and the proportion of attacks that go undetected is known as the *false negative rate*.

The advantage of a threshold scheme is it's simplicity, and in fact simple threshold schemes are still the most widespread in commercial practice. The meaning of a threshold is readily understood by both IT staff and users. However, more complex schemes have been developed, and to those we now turn as we begin to discuss statistical profiles.

The "number of login attempts" variable that we have been studying is an example of what is known in anomaly detection as a *measure*. Statistical anomaly systems frequently have large numbers of measures designed to try to capture different possible things that could go wrong with the system. Some statistical algorithm is applied to each of the measures. We could look for people trying to login to too many different systems, users who look at too many different files, users who login at unusual times, users who come from strange places, etc, etc.

A drawback of threshold schemes is that they expose this complexity (large numbers of measures) to the user of the system. While any individual threshold is easy to understand, if the system has many thresholds understanding them well enough to adjust them may be a prohibitive task. This is particularly true in situations in which the proper value of the threshold will vary from one site to another, and has motivated the search for techniques that will adaptively learn, rather than rely on the software developer to guess the right threshold for the users deployment, and then the user to figure out how to fix it when it is wrong.

Several properties of statistical distributions are important. One is the mean (sometimes informally referred to as the average). The mean is the "center of gravity" of the distribution – if the bars in the histogram were all made of some solid material, the mean is the point on the x-axis where the distribution would teeter at balance, rather than immediately topple to left or right. The mean is the average of all the observations.

Other important concepts are the variance and the standard deviation. The standard deviation is a measure of the average width of a distribution – how far, on average, are the data observations from their mean[10]. Very scattered observations will have a large standard deviation, while tightly clustered ones will have a small standard deviation.

The variance is just the square of the standard deviation.

The standard deviation, often referred to by *sigma* the name of the Greek letter used to denote it, is sometimes used as a different way to assess the anomalousness of an observation. The more "sigmas" an observation is from the mean, the more significant it is taken to be. For example, "two sigmas" is the bare minimum to be noteworthy at all, but three or four sigmas are much better evidence of an anomaly.

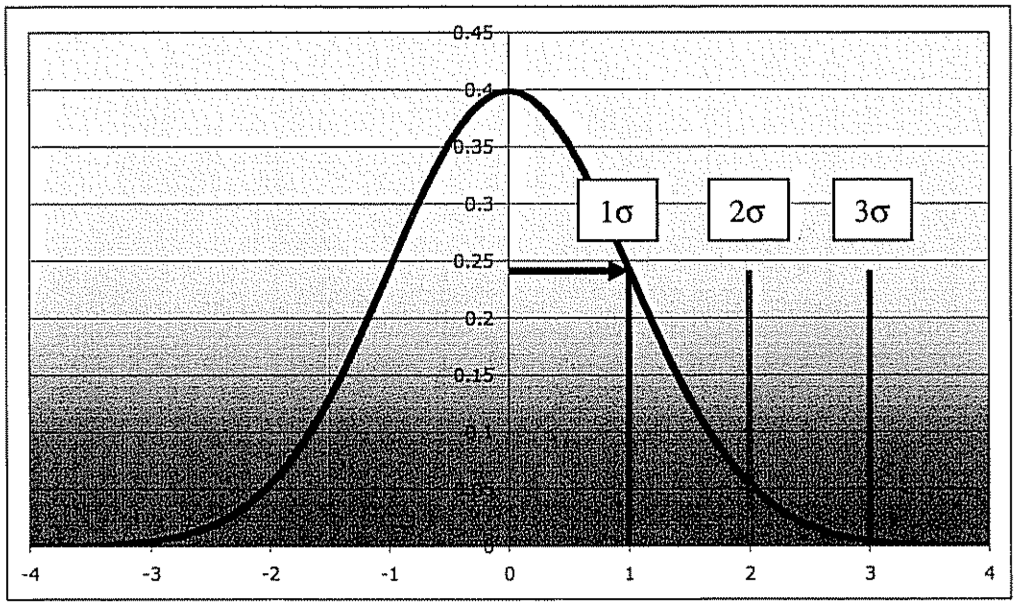


Figure 17: Standard normal distribution. Length of arrow shows the size of the standard deviation. One, two, and three sigmas from mean are shown as black, plum, and blue

[10] Technically, the standard deviation is the root-mean-square distance of the observations from the mean.

This is illustrated above in the case of the *normal distribution,* a mathematically important probability distribution[11]. The mean is at the center of the peak, and the 1σ, 2σ, and 3σ points are shown. Clearly, quite a lot of the distribution is more than one standard deviation from the mean (about 1/3 of the area under the curve). Only about 5% is more than two standard deviations from the mean, and less than 1% is more than 3σ from the mean of a normal distribution (these percentages are properties of this distribution, and would be different in other distributions). Thus for a normally distributed variable, a three sigma anomaly is quite significant.

The normal distribution is an example of a *parametric distribution*. This term refers to a probability distribution that can be described by a closed mathematical expression which accurately captures the distribution. A large body of mathematical techniques are results are available for performing statistical tests for in *parametric statistics*. In the early years of computer intrusion detection, people tried to apply this theory. In general, the goal of building a *statistical profile* is a definition of measures to be studied and some statistical/probabilistic summary that will allow for an efficient determination of whether new observations are anomalies or not. A simple profile would be just to store the mean and standard deviation of the data and then determine how many deviations from the mean new observations are, setting a threshold on that. It has since been realized that parametric distributions are generally a poor representation of the kinds of measures used in this field, and non-parametric methods have become more common. Specifically, measures in networking and network security are often *heavy-tailed* – they fall away to nothing much more slowly than a normal distribution.

Another approach to statistical profiling would be to store an explicit representation of the probability distribution of all observations seen to date. Given a new observation, this would allow a determination of how probable such an observation was (based on how much of the historical probability distribution of the measure was further out in the tail than the latest observation). This unfortunately has drawbacks also. Specifically, measures in realistic situations often have *non-stationary* properties. What this means is that the probability distribution of observations is not fixed over time, but changes for various reasons. This can cause inferences based on historical measurements to be wildly wrong when the distribution changes, resulting in problems of high false-positives, false-negatives, or both.

Thus, an effective statistical profile should learn the changing distribution that the measure is following. This was the motivation for some of the early research in statistical anomaly detection which we will discuss in a later section.

IIc) Background on Computer Security.

For the first three decades of computing, from the development of the ENIAC computer for use in calculating the path of artillery shells in 1945 to roughly the late 1960s, computers were large specialized installations that ran programs delivered to them by direct physical input (eg by feeding stacks of punched cards to a reader). Such computers

[11] Also known as a *Gaussian distribution* in some literature.

were not connected to any networks, phone lines, or even terminals. As such, security threats to them were very similar to security threats to files or documents; the computer and its sensitive data could only be compromised by obtaining physical access to them. Since there was no specialized threat to computers, there was no specialized field of study called computer security.

This began to change in the late 1960s with the advent of time-sharing computers which allowed multiple users to run programs on the system at the same time (beginning with the Dartmouth Time Sharing System in 1964), with the system switching between users rapidly and presenting the illusion to all of simultaneous access (typically via terminals which might be some distance from the computer itself). Such systems allowed each user to have their own storage files and parts of the computer memory. Thus there was the potential for users to read each others files, write over each others files, or for their programs to interfere with each others operation. Thus the field of computer security was born: mechanisms were needed to ensure that users could not interfere with each other.

The early focus of computer security, starting in the late 1970s and early 1980s, was figuring out what kind of security policies the computer should apply, and then attempting to find ways to ensure that the design and implementation of the computer, especially the operating system, could guarantee that the policies would invariably be followed. Much of this work was inspired by the US Department of Defense (DoD) which had a desire to be able to use computers at multiple classification levels while enforcing a policy that would ensure people and processes without secret clearances couldn't access secret computer data, people and processes with only secret clearances couldn't access top-secret data, and so forth.

Researchers quickly realized that a major problem with any kind of access control policy was ensuring that the system actually behaved according to whatever rules its designers and owners had tried to imbue it with. It turned out that there was a huge problem with subtle flaws in the design or implementation that led to ways to circumvent the protection mechanisms. Hackers and vulnerability researchers developed increasingly (and incredibly) ingenious methods for circumventing the rules. A flaw that has the potential to allow the security rules to be circumvented is known as a *vulnerability*, while a technique for taking advantage of a vulnerability is an *exploit*.

For example, one common class of vulnerabilities is called a *buffer overflow*. In these cases, a program is reading in some input (eg from the keyboard) and placing the characters in a region of memory called a buffer. If the programmer makes a mistake in the way this is done and fails to check how much input is being supplied, the program will work normally under almost all circumstances (meaning the error will probably not be found in normal testing). However, an attacker may be able to feed the program specially crafted overlong input which will be written to areas of memory adjacent to the buffer that were holding other critical data that the program was using to keep track of where it was up to in its operations. By supplying carefully crafted input, the attacker can overwrite this status data and cause the program to begin executing new instructions in the computer memory that were supplied by the attacker. Once this has been achieved, the program is then effectively entirely under the control of the attacker and security has been circumvented.

The realization of this kind of problem led to a huge focus on how to verify the correctness of programs and computers as they were being designed and built. This culminated in the Department of Defense "Orange Book" [DoD85]. In addition to mandating features (such as audit trails) that the computer system must have, the rules also covered a variety of processes for reviewing design and implementation to ensure there were no errors that could lead to security problems. The DoD attempted to impose a requirement that computer systems it purchased must conform to Orange Book standards in an attempt to influence the marketplace to adopt more secure computers.

However, this failed. It turned out to be much more time-consuming and expensive to develop systems in a secure manner, so that insecure computers were faster and better in other ways by the time computers designed according to Orange book standards reached the market. Customers preferred having speed and features to having security. Security problems have continued to dog all major deployed brands of computers ever since.

The problems were illustrated through a series of famous hacking cases in the 1980s and 1990s. One example will suffice for here. A young astronomer, Clifford Stoll, was asked to resolve a 75c accounting error in the computer systems at Lawrence Berkeley Lab (one of the Department of Energy's national laboratories). This led to the realization that a hacker had penetrated the computer system. Stoll became obsessed with tracking him down, and eventually was able to find which telephone line the hacker was using so that Stoll could keep a permanent eye on his activities. The hacker was clearly searching for data containing sensitive terms related to the Strategic Defense Initiative (SDI: an 1980s era missile defense program). Stoll concocted an elaborate scheme to create a fake respository of SDI information to cause the hacker to stay on the system long enough to be traced back to his origin. The intrusions turned out to be coming from a German hacker called Marcus Hess, who was selling the information he obtained to the KGB.

A technique that hackers use that is of particular importance to us here is what is today usually known as a *scan*, though in the mid-1990s, the terms *sweep* or *doorknob rattling* where often used for the same general class of activity. The idea of a scan is that an attacker might not know much about the computers on a given network when he began the project of intruding on that network. Thus his first goal would be to perform reconnaissance, and in particular to establish what computers were on the network, and which ones might be vulnerable to means of exploitation available to him. To this end, he will cause his attacking computer to attempt to contact sequentially a number of possible targets and in some manner probe them. In early examples, when guest accounts with default passwords were common, hackers would scan the machines on the network and in each case they would connect to it via telnet and attempt to login using various popular default login and passwords. Sometimes the scanning is systematic – every IP address on the network will systematically be contacted. In other cases it's random. Typically, a program is used to automate the scanning process.

Another development that galvanized the development of computer security was the Internet worm incident of 1988. In this incident, a young Cornell graduate student named Robert Morris released a *worm* – a self-propagating program that broke into computers via one of several exploits it carried and then ran itself on this computers and repeated the process again.

One of the outcomes of the Internet worm incident of 1988 was the funding by the federal government of a national Computer Emergency Response Team (CERT) as a clearing house for information about computer vulnerabilities and incidents. By 1995 when formal statistics on vulnerability reporting began, CERT was seeing 171 vulnerabilities. This has grown by leaps and bounds, with the 2005 number being 5990. The computer systems continue to have very large numbers of vulnerabilities.

It was in response to this situation in the late 1970s and 1980s that the need for computer intrusion detection became clear. It was discovered in practice that it was extremely difficult to build perfect security into marketable computer systems, and it was already obvious that intruders were going to cause serious harm by exploiting the resulting vulnerabilities.

IId) Background on Intrusion Detection

The first paper to discuss the possibility of automated computer intrusion detection was a report by James Anderson in 1980 [And80] for the US government. Anderson considered the problem of manual audit trail inspection which was then being performed for audit trails of an IBM mainframe used for classified computing projects to ensure the security of the computer. This was extremely time consuming. Anderson's paper discusses at length the possible risks to the computer, the deficiencies of the audit trail for detecting attacks on the computer, together with various possible improvements.

Anderson also considered the possibility of automated statistical detection of computer intruders masquerading as other users. He considered such measures as what time of day a given user was accessing the computer, how much resource usage (cpu etc) a given program was using, and which files a user or program was accessing. He postulated finding the average and standard deviations for these various measures, and looking for situations where a user or program was too many standard deviations from the mean of such program executions. He also considered comparing measures against fixed thresholds. Anderson's report contemplated a batch system that would run nightly to process that days' security audit trail.

The Anderson report was an analysis of requirements and some initial design ideas for an audit monitoring system. Such a system had not been built at the time.

The next significant development in intrusion detection research was made by SRI, and SRI's early contributions to the field are worthy of their own section.

IIe) Early Intrusion Detection Research at SRI

Here we look at what SRI developed between the early 1980s when they began work in this area, and the end of the NIDES project in 1995, the last day before the statutory bar for the first of the patents that they filed. All work described here is published prior art for the purposes of SRI's patents. For the sake of clarity, we consider the development of SRI's systems in isolation, but readers should be aware that significant developments were made by other research groups also, and some of those will be described later in this document.

SRI's research began in the early 1980s with an investigation for the US government of audit records on an IBM mainframe similar to the one considered by Anderson [SRI06]. However, it was not until 1986 that Dorothy Denning published a paper on the IDES (Intrusion Detection Expert System) model. That paper has gone on to be the most widely cited paper ever in the field of computer intrusion detection, with over 800 citations to date. The paper describes an outline design for an intrusion detection system which would operate to detect computer intrusions by inspecting computer audit trails in real-time.

The paper discusses an abstract model for what security audit trails should look like (together with practical deficiencies of various audit trails of the time such as the SMF facility on IBM mainframes, and the recording facilities of Berkeley Unix). An idealized audit trail has a *Subject* (the user or program performing an action), an *Object* (the file, program, etc that was being acted on), the *Action* that was being taken by the subject on the object (essentially equivalent to the system call that was being invoked by the subject program), an *Exception-Condition* field which recorded any errors that might have occurred in carrying out the action, a *Resource-Usage* record of how much computer resources were required to carry out the action, and a *Time-stamp* which records when exactly the action occurred.

Given a stream of such audit records, the IDES paper contemplates various kinds of statistical metrics (alternatively called measures in the paper), including *event counters* which count the instances of some type of audit record, such as "the number of logins during an hour, number of times some command is executed during a login session, and number of password failures during a minute". Then there were *interval timers* which represent the length of time between two related events, and *resource measures* which consider the amount of resources used in some action during a period. Examples "are the total number of pages printed by a user per day and total amount of CPU time consumed by some program during a single execution"

Given a measure constructed out of audit records, the paper then contemplates a variety of *statistical models* for this model. For example, the paper refers to the possibility of comparing the metric to fixed thresholds (as foreshadowed in Anderson's paper), comparing an observation to the mean of past observations of that metric to see whether it was too many standard deviations away (also mentioned in Anderson). However, the IDES implementation seems [Jav91] to have ended up by focusing on using a multivariate approach in which the a short term exponentially aged average of the entire vector of intrusion detection measures was compared to a longer term covariance matrix of past behavior of those measure.

In IDES, a statistical profile consists of a definition of the kind of audit record to be considered, the metric to be used, and the type of model to be applied to determine if a new observation was anomalous relative to the past behavior of the system. A variety of possible profiles based on system audit trails are laid out.

Additionally, IDES had a rule-based component allowing it to look for hard-and-fast scenarios which reliably indicated intrusions.

In the early 1990s, IDES was re-engineered by SRI and renamed NIDES, for Next-Generation Intrusion Detection Expert System. To a significant degree, the changes from NIDES to IDES were engineering and practicality improvements rather than fundamentally new ideas. For example the system now acquired a graphical user interface. The statistical algorithms were changed slightly to make them more robust to realistic distributions of activity.

However, one development of significance to this case is that NIDES was designed to run on a network of computers that were interconnected to one another. The system still worked by examining the audit trails of individual computers, but it now had a client-server architecture, allowing multiple audit trails from multiple computers to be consolidated together and processed. Some of the rules in the rule base began to reason about network activity.

For example ([And94 p89-92]), NIDES in 1994 had rules to detect a variety of network services being used on the machine being audited. Thus if a user made a connection to the login, or remote shell, or mount service, this fact would be noted by NIDES. In some cases, NIDES distinguished between successful connections and unsuccessful connections (eg the RemoteMount1 rule reported successful remote mounting of a file system, while the RemoteMount2 reported cases where the remote mounting failed due to some kind of error).

Further development of NIDES in this direction was contemplated. Eg in [And95], consideration of NIDES being extended to directly read network packets was discussed as a future possibility. The packets would be converted to NIDES audit records in their canonical format. Consideration was given to what kind of statistical measures might be applied to the network records:

> "For the NIDES statistical component, a set of statistical intrusion-detection measures that are specific to network-level intrusion are easily developed. For example, such measures might profile the amount of traffic originating or being received by local and remote hosts for given times of the day and days of the week. This information could be correlated with information received from audit trails, such as port number and network activity type, to build additional measures of interest."

In fact, there was to be a funding changover for SRI's intrusion detection efforts from the Navy to DARPA. There were also some personnel changes at SRI – leaders of the IDES/NIDES development such as Teresa Lunt and Debra Anderson were to leave, and Phillip Porras joined the group. When funding for SRI's intrusion research resumed, under Mr Porras's leadership, the group began another cycle of re-engineering the system, which was now called Emerald (Event Monitoring Enabling Responses to Anomalous Live Disturbances), which we shall discuss later.

For convenience, here is a timeline of the major early research systems developed by SRI and UC Davis.

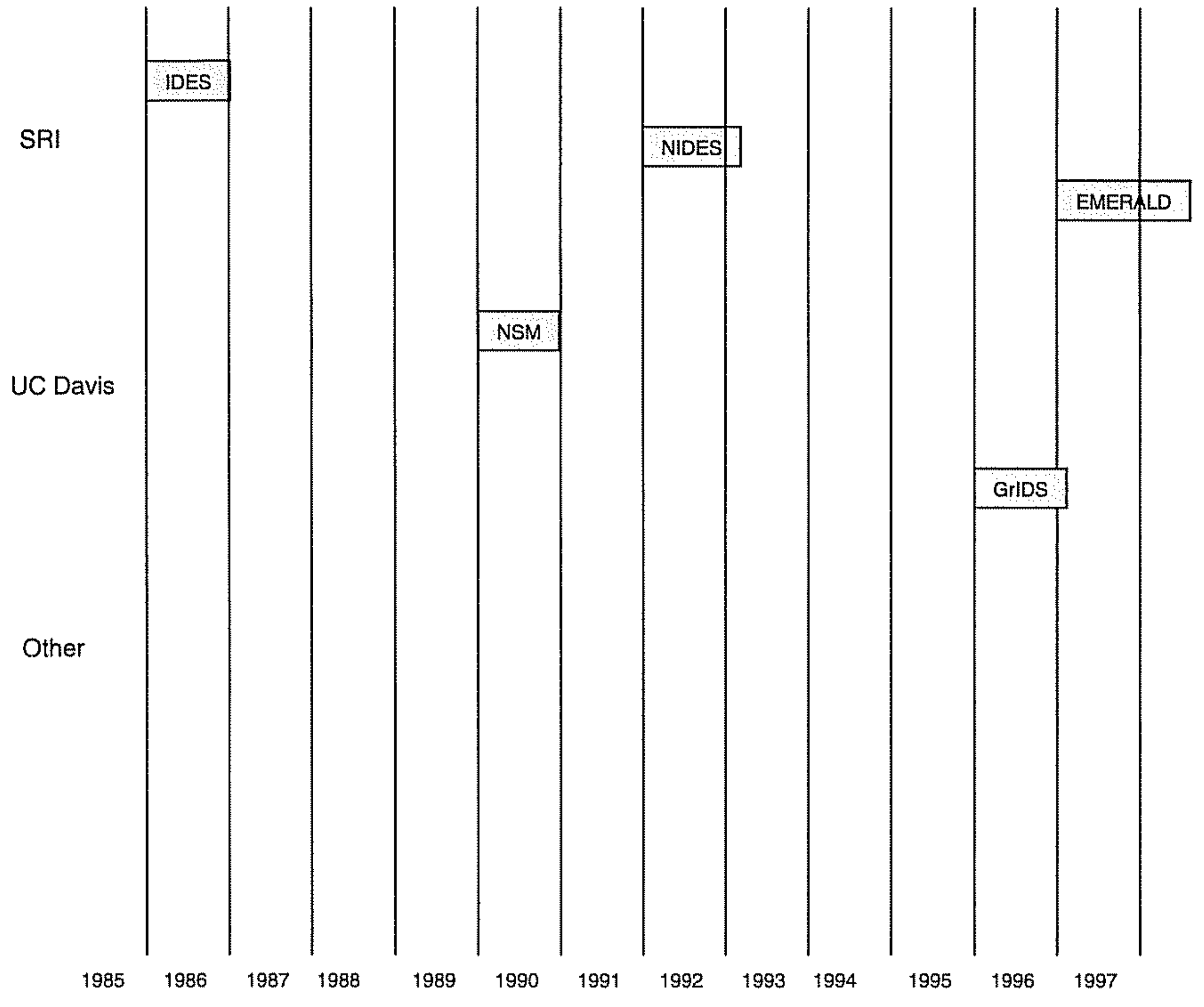


Figure 18: Timeline of major intrusion detection systems discussed in this report

IIf) The NIDES Statistical Algorithm

Because it will be important later, we give here a detailed description of the NIDES algorithm. This builds on our introduction to statistical anomaly detection, and will be important to understanding the issues later. NIDES was more-or-less the state-of-the-art in statistical anomaly detection for detecting intrusions prior to the statutory bar for these patents.

The NIDES statistical algorithm is quite complex and we shall work up to it in several stages. We will start by considering the case of a single measure. For our measure in explaining NIDES we will choose a hypothetical example. Imagine that every time the user on a modern Windows system clicks to change from one application to another, or starts the computer and selects an application, we create an audit record of what the newly selected application was, along with the time of selection and the user doing the selection. Clearly, this data would tell us something about what applications a user makes use of, which in turn would be dependent on their work and leisure activities on their computer.

One user, let us call her Alice, might be a retired homemaker who mainly uses her computer for email with her family, and to check things on the web. As such, almost all of her computer use is in just two applications: Outlook and Internet Explorer (IE). A second user, let us call her Brenda, is a financial analyst and uses a mixture of office applications (Excel, Word, Powerpoint), along with Outlook and IE. She also uses her computer to download music with iTunes.

This kind of measure is called a *categorical measure*, since the variable is drawn from a list of categories (in this case computer applications), rather than being a number.

If we took this measure of the two users averaged over a long period, we might find histograms something like the one here in Figure 19.

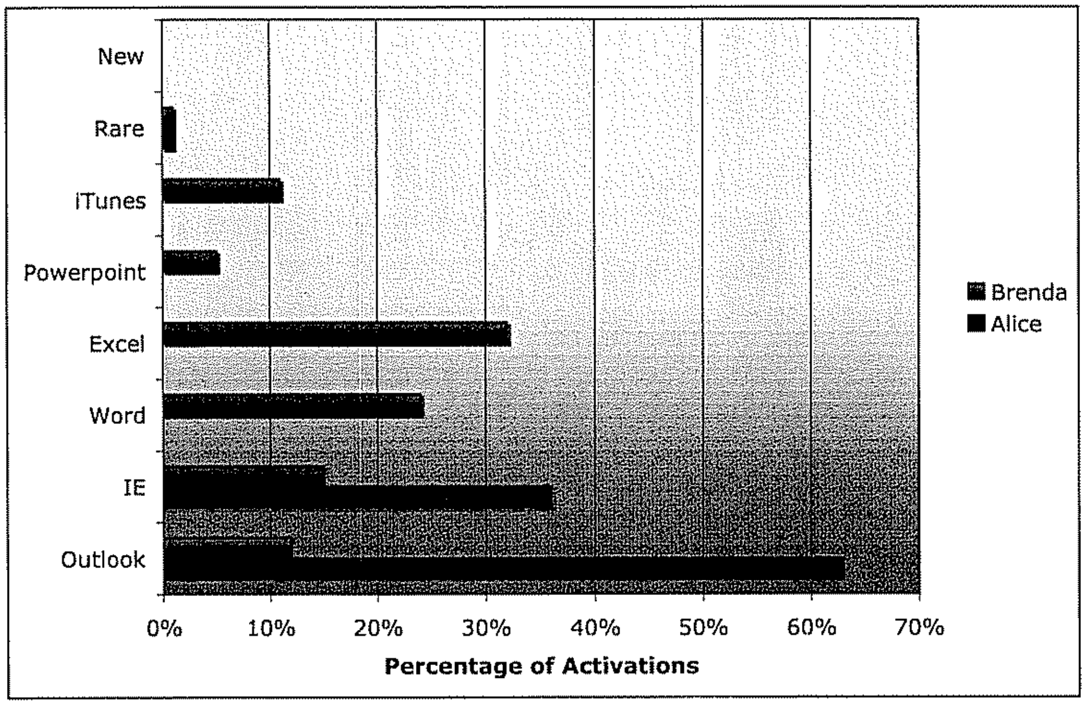


Figure 19: Long term averages for application choices by two hypothetical users

This is a way of representing the *long-term-profile* for these users, or at least that part of the profile associated with this particular measure (Application Activations) – the full profile will involve multiple measures. Clearly, Brenda's profile looks quite different than Alice's. Brenda uses applications that Alice never uses. Alice very occasionally uses some other applications that get lumped together in NIDES into a *rare category*, but neither user ever uses a new application.

On the other hand, if we checked over a shorter period, perhaps about a day, we would find somewhat different frequencies of application usage. Perhaps in this particular period, Brenda was preparing for an important presentation and using PowerPoint much more heavily than on average.

The next graph in Figure 20 adds this particular day of Brenda's activity (her *short-term profile)*, along with her long-term and Alice's.

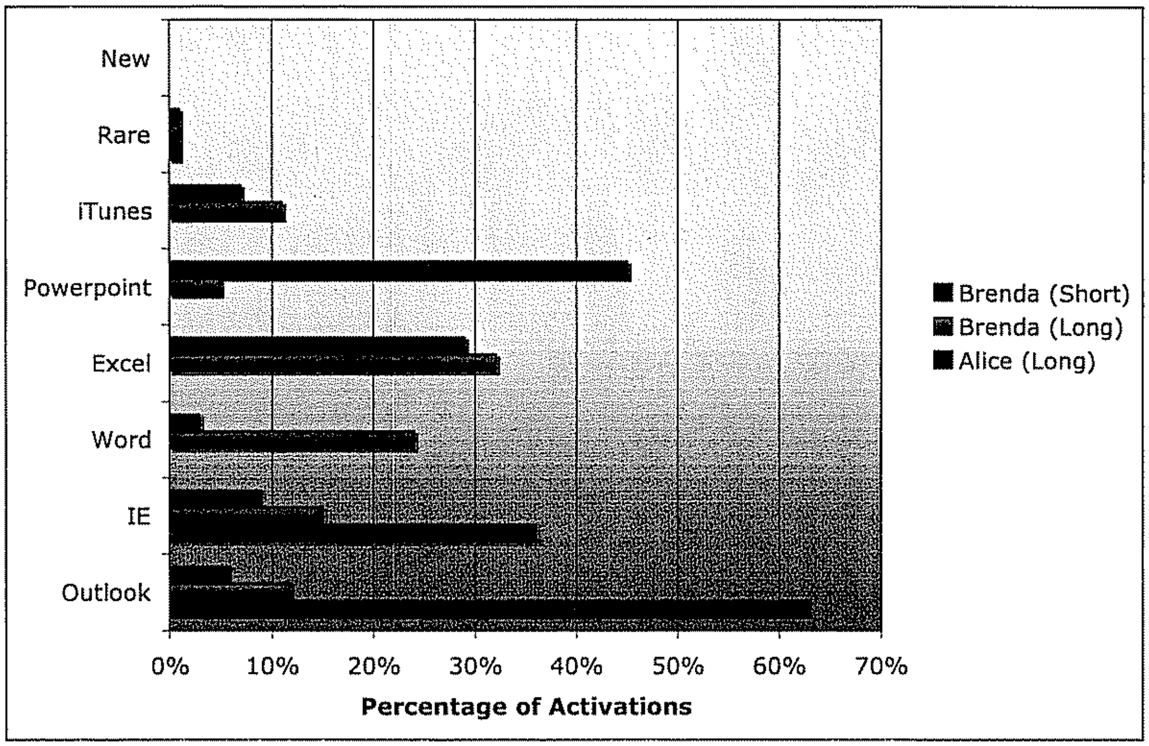


Figure 20: Long term application choice profiles for Alice and Brenda, with short term profile for Alice over some hypothetical period.

Clearly, although Brenda's short-term profile (the blue bars) is different than her long term profile (the plum bars), it still looks more like Brenda than Alice (the green bars). The basic idea of the NIDES statistical algorithm is to develop a way to compare these short-term profiles to the long-term profiles. The hope is to be able to distinguish users from one another, such that, if Jack, an industrial spy, slips into Brenda's office at lunch time and starts using an archive utility to burn DVDs of Brenda's files, while downloading, compiling, and running a backdoor program that will monitor Brenda's keystrokes, these unusual program invocations will change Brenda's short term profile enough that an intrusion detection system can detect that someone is *masquerading* as Brenda.[12]

Let us now explore in more detail how NIDES built statistical profiles and compared them, starting with the long term profile.

The long-term profile is updated every night, and involves incorporating the distribution of categories that occurred during that day. The way this is done is via a technique called *exponential averaging* that allows efficient incorporation of new information into a running average without keeping the entire history around to recompute the average.

[12] Note that the NIDES algorithm has not achieved widespread commercial success for, but it was quite influential in the research community.

Exponential averaging also causes profiles to adapt smoothly and gradually to changes in user behavior.

The idea is that we take the old profile, multiply all the columns down by a value that is a bit less than one, take the days histogram, multiply it by one less that value, and then add the two. This results of an average of the old information with the new information, the process is illustrated in the following graphs. First we show the long-term profile, together with the new day's data.

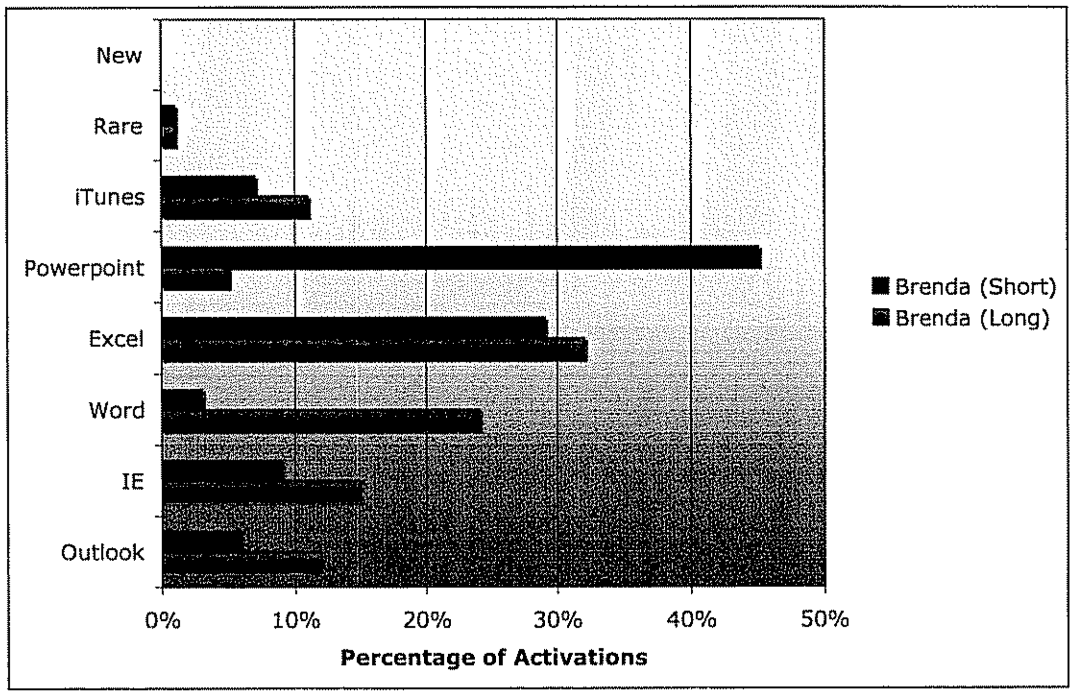


Figure 21: Long term profile for Brenda, together with information at the end of the day ready for incorporation with long-term profile

Next, we show what happens after both profiles have been scaled down. In this example, we multiply the new profile by 1/8, and the old profile by 7/8:

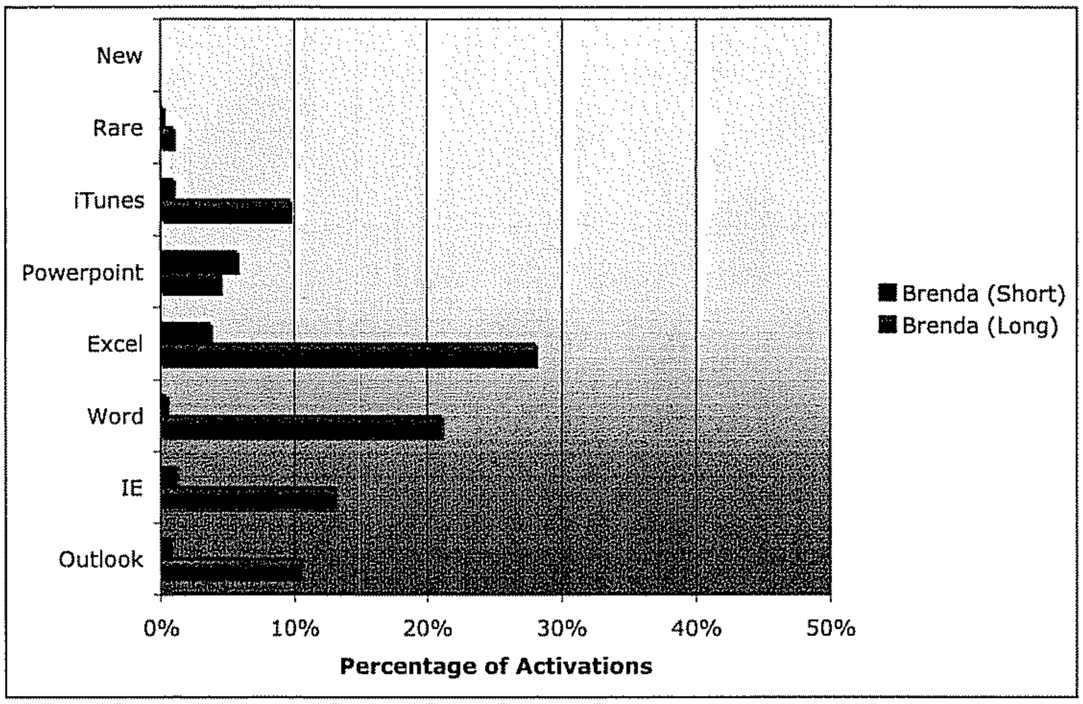


Figure 22: Long term profile for Brenda, together with information at the end of the day ready for incorporation with long-term profile. The long-term has been scaled down by 7/8, and the short-term by 1/8.

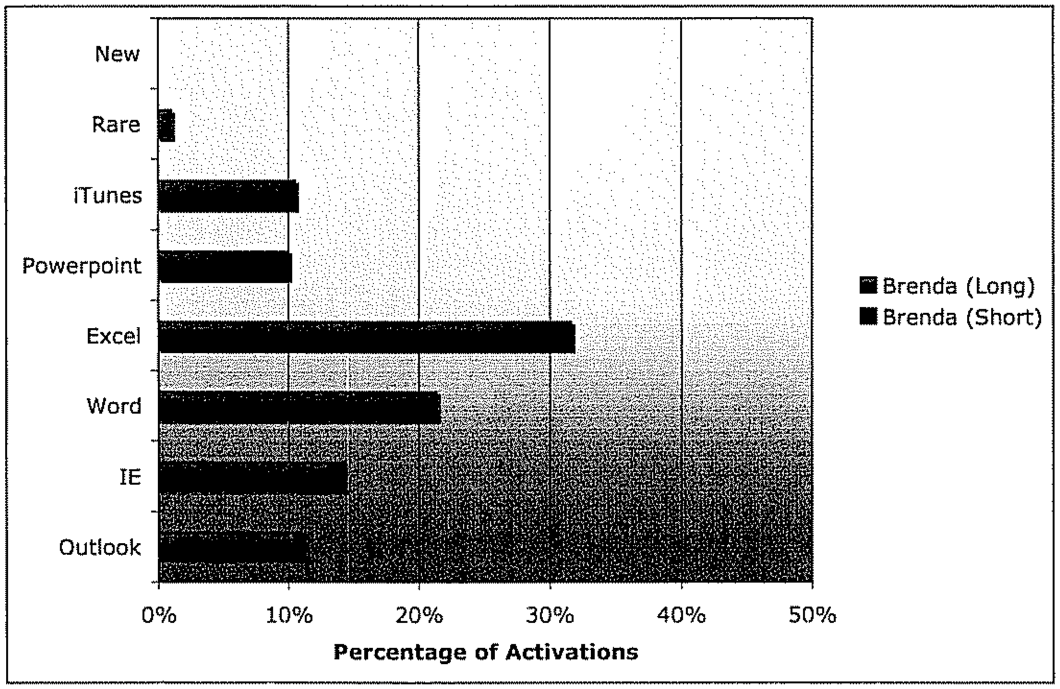


Figure 23: The new long-term profile showing the contribution of the old long-term profile added to the contribution from the most recent day

Note the way in which the very strong representation of PowerPoint in the recent day has had the effect of increasing PowerPoint in the long-term profile. If Brenda, having finished her presentation, did not use PowerPoint again for a few days, the value of PowerPoint in the long-term profile would slowly decay until the next time she worked on a presentation.

If the reader thinks about the influence of any given day in the long-term profile, it is at its greatest immediately after it has been incorporated. The next night, that particular day's contributions will, like all of the long-term profile, get multiplied by 7/8 in order to combine in the latest data. Each subsequent night, another muliplication with 7/8 will occur. After approximately five nights, the day in question will have had it's influence reduced by half. After 10 nights, it will be down to 1/8, and after 15 nights, 1/8. Clearly, days from long ago will have hardly any influence on the current long-term profile. Exponential aging forgets old data, and overweights the newest data. This allows it to adapt to changes in the profile. The time required for the influence of a given contribution to fall to half is known as the *half-life* of the exponential aging process. In this example, we make it about five days for illustration, but the NIDES documents record a half life for the long-term profile of 30 days. Thus it would take NIDES several months to forget things that had happened.

The short-term profile is computed in a basically similar manner, except it is updated after every audit record instead of every day. Thus it responds much more rapidly to changes in the user's behavior. Note that the short-term profile is different than the accumulated days statistics used to update the long term profile in the way NIDES was designed. The function of the short-term profile is be compared against the long-profile to see if the two are becoming very different.

Specifically, the first thing that is computed in differencing the short-term profile and the long-term profile is what NIDES refers to as the *Q-statistic*. The Q-statistic is computed by taking the difference in the heights of the bars of the short-term profile and the long-term profile. Those differences are scaled down by the average variance in the short term profile bars. They are then summed and squared. This gives a metric which will be zero when the short-term profile and the long-term profile are of identical shapes, and will get bigger the more the two have a different shape. The process is illustrated via the following figure.

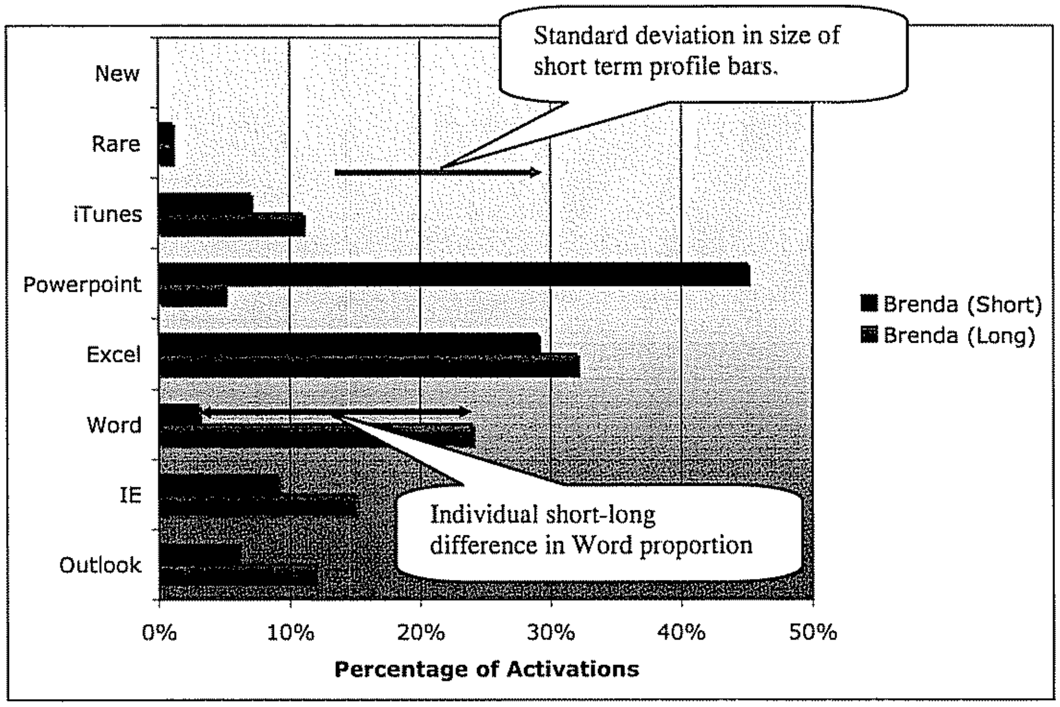


Figure 24: The process of computing the Q statistic of the difference between short term and long term profiles. The individual differences (black arrow) are each computed, and then divided by the variance in the short term profile frequencies (red arrow).

However, NIDES does not use Q directly. To understand why, consider that different users may vary widely in how much their short-term profile fluctuates from the long-term one. Imagine comparing Alice, our retiree earlier, with Carl, who's job is to test new applications before they are deployed on his employer's network. Alice uses almost exclusively two applications, and only the relative frequency of her use of them varies, and that mildly. By contrast, Carl is constantly using new applications, and discarding the use of ones he tested earlier. Carl's profile is constantly shifting and unstable, and his long-term profile is invariably struggling to catch up with his short-term profile from a great distance.

To handle this, NIDES adds an extra layer of transformations onto the Q variable. The first thing they do is maintain a histogram over time of the values the histogram takes on. Thus they keep track of the fact that Alice's Q is usually in a narrow peak close to zero, while Carl's Q is never very close to zero, and is spread out over a large number of variables.

Then they construct a new variable S, which is analogous to Q, - it is big when Q is big and small when Q is small. However, it is explicitly constructed to have the standard normal distribution reviewed earlier. This is a non-parametric technique – Q has an unknown distribution, but by tracking it explicitly in detail via histograms, it is possible to create an S variable which has a well-known and simple distribution.

The final step in the NIDES algorithm is to note that the S for the Application measure is just for a single measure. NIDES and its derivatives were intended to work on many measures at the same time. The way they are combined is simply to take a special kind of average of all the S statistics (the root mean square), and thus create what NIDES calls the T^2 statistic. T^2 effectively summarizes the combined total anomaly of the short-term profiles of all measures compared to their long-term profiles. T^2 by construction has a well-known parametric distribution called the χ^2 distribution. This means it is straightforward to assess whether or not T^2 is anomalous, and how statistically significant the anomaly is, if there is one.

Whether the anomaly has anything to do with a security violation or simply reflects a user changing their behavior is another question again.

The "application usage" measure we just discussed is a type of what NIDES calls a *categorical measure*. This means a measure that discusses the relative frequency with which a variable takes on one of a number of possible distinct values. NIDES considers several other kinds of measures:

- *Intensity measures* look at how often audit records (or events or packets) arrive at the system
- The *audit record distribution* measure is a special case of a categorical measure that explores the distribution of different types of audit records (or events).
- *Counting measures* explore the distribution of a numerical value of some kind (such as the amount of cpu used by a computer, or the number of bytes of data in a connection).

That completes our discussion of the NIDES algorithm. Some detail and some cases have been left out for the sake of brevity, but this description should suffice to illustrate the issues in this case.

IIg) Intrusion Detection Research at UC Davis

The intrusion detection group at UC Davis was founded by Professors Karl Levitt (who retired from SRI and took a faculty position with UC Davis), and Biswanath Mukherjee (a Networking research professor). Together with their students, particularly L. Todd Heberlein, they did some of the most important early research in intrusion detection after SRI.

IIg1) Network Security Monitor (NSM)

The first and most influential system that UC Davis developed was the Network Security Monitor (NSM), developed beginning in the late 1980s, and described in publications from 1990 onwards [Heb90, Heb91a, Heb91b, Heb91c, Muk94, Heb95]. The NSM was the first *Network Intrusion Detection System (NIDS)*. The *Network* in NSM refers to the fact that the system worked by directly inspecting network packets as they were transmitted on a network wire between different computers, rather than inspecting activity on one or more computer systems by looking at their audit trails. All previous systems worked from audit trails.

Operating from network packets turned out to have considerable practical advantages over working from host audit trails, and this style of intrusion detection has proven more important in commercial practice, and spawned many commercial product versions of basically the same idea. The ability to monitor the network allows NIDS systems to be installed at network choke points where the behavior of large numbers of computers can be conveniently monitored from a small number of locations. Additionally, there is no resource impact on the computers being monitored (though the computers or other devices doing the monitoring may need considerable resources). These are practical and administrative advantages that the marketplace has found compelling.

The NSM itself was deployed initially at UC Davis, and the published papers provide a variety of operational experiences there detecting actual intrusion, and some quantitative evaluation of the system. Variants of the same software went on to be widely deployed at US Department of Energy and US military computer installations for the better part of two decades. In the mid to late 1990s, almost the entire US Air Force unclassified network was monitored by ASIM – a descendant of the original NSM. NSM worked exclusively from TCP/IP packets taken from Ethernet networks.

The NSM employed a mixture of statistical and rule based techniques to decide whether or not a given piece of network activity was suspicious. The algorithms were generally inspired by IDES (the original IDES paper was published in 1986), but differed considerably in detail. NSM also discloses several different approaches to the problem that are important to this case, and which I will describe. NSM predates NIDES and thus was not influenced by that later system.

NSM built a warning score that was a weighted sum of three components. There were:

- A statistical anomaly detection algorithm that looked for strange connections and traffic mixes.
- A rule-based system that looked for known bad patterns of traffic.
- A heuristic that assessed the security of endpoints and assessed an intrusion to be more likely if an insecure system was putatively attacking a secure system than the other way around.

The NSM relied heavily on the concepts of *traffic matrices* and *traffic masks*, which I will summarize here. A *matrix* is a mathematical construct that is similar to what is informally known as a table. NSM used a four dimensional matrix, but we shall introduce the concept with a two dimensional simplification, and then work from there.

The dimensions that NSM worked with were the source IP address (of a packet or stream), the destination IP address (of the same), the service (destination port), and a *connection-id* (which was a specially constructed variable designed to ensure that every connection had a different value. Let us start just by working with the source and destination address. Suppose there were four hypothetical computers on the network, which we shall label ".1", ".2", ".3", and ".4" (this denote the different IP addresses of the computers. Let us construct a table with the odds that each of these computers will talk to one another on any given day. It might look like the following:

		Destination			
		.1	.2	.3	.4
Source	.1	0%	100%	96%	68%
	.2	100%	0%	0%	0%
	.3	96%	0%	0%	54%
	.4	68%	0%	54%	0%

Table 2: Traffic mask for hypothetical small network

The way to read this table is that the source labels the rows and the destination labels the columns. The percentage in each cell is the probability (odds) that a given source will contact a given destination on a given day. Thus, for example, this traffic mask estimates there is a 68% chance that the computer with address “.4” will source packets to “.1`” on a randomly chosen day (the lower left cell in the table.. In this example, all the data-flows are symmetric so the upper right cell, the probability that “.1” will source packets to “.4” is also 68%. The diagonal elements of the table are the probabilities that hosts will send packets to themselves. Since these would not appear on the network but be delivered internally, those probabilities are zero. In our example, all the computers talk to .1 with some frequency, but other than that, only .3 and .4 talk. In realistic examples on larger networks, only a small proportion of the possible conversations actually have significant probability of occurring – most computers do not talk to most other computers (just as in a large office, most people do not talk to most other people but only to a smaller circle of coworkers.

Thus if on some occasion .2 suddenly began talking to .4, we could say that this was anomalous. Of course in a complex and changing network, a single anomaly like this might well just represent an innocent change in behavior. However, a large number of anomalies would be likely to represent either an intrusion, or some kind of network or computer problem causing network traffic to be unusual. This was the basic idea behind the statistical approach of the NSM.

Now, to get closer to what the NSM was doing, we need to complicate the picture more. If the reader will imagine taking the two dimensional table above and stacking many instances of them on top of each of other to gain a three dimensional table, then each layer in that three dimensional table could be indexed by the service (or destination port, such as for the world wide web (HTTP), email (SMTP), and so on. Thus each cell in the three dimensional matrix corresponds to the possibility of connections from a particular source to a particular service on a particular destination. Now if the reader imagines taking a large stack of these three dimensional tables and spreading them out (along some admittedly hard-to-visualize fourth dimension), where each cell in the fourth dimension corresponds to different occasions when there was a connection between that source and destination on that particular service (there could be many such connections over the course of the hours and days of operation of the network), then taking each cell in the four dimensional matrix and storing the number of packets and bytes associated with that direction of the connection, then the reader will have some idea of what the NSM meant by a *traffic matrix*.

A *traffic mask* was similar, except that instead of storing information about each connection that occurred, it stored a statistical description of the probability of a connection occurring for a given source, destination, and service combination, and it also stored an estimate of the probability distribution of a given number of packets being in the connection, and a given number of bytes being transmitted in the connection.

Every five minutes, the traffic matrix of current traffic was compared to the traffic mask, and the anomalousness of the matrix was computed. Every 15 minutes the current traffic mask was saved to disk. At long intervals, the traffic mask of normal traffic was computed from a large set of the current traffic matrices that had been stored every 15 minutes.

The probabilities of connections occurring were estimated using the same exponential aging approach described in the NIDES algorithm [Heb91a].

In addition to the statistical approach, NSM also implemented some fixed rules including

- Sources connecting to more than 15 destinations were suspicious
- Sources trying to login more than 15 times were suspicious
- Sources trying to connect to non-existent destinations were suspicious.

IIg2) Distributed Intrusion Detection System (DIDS)

After the NSM, the next major system that UC Davis developed was DIDS. DIDS was deployed at UC Davis, and subsequently was re-engineered by Trident Data Systems and deployed at least at test sites at the US Air Force. The UC Davis version of DIDS, described in a series of publications [DIDS, DIDS-Feb, DRA, SNA 91] was a prototype intended to research the issues with integrating and correlating information from a variety of sources in a computer network to come up with a coherent single picture of distributed intrusive activity. DIDS was the first system to use the architecture that has since become widespread commercial practice in the industry in which a set of sensors examine data from hosts and networks, perform some detection and data reduction locally, and then feed activity to a centralized management platform which does further correlation and detection and presents a coherent picture to users of the system.

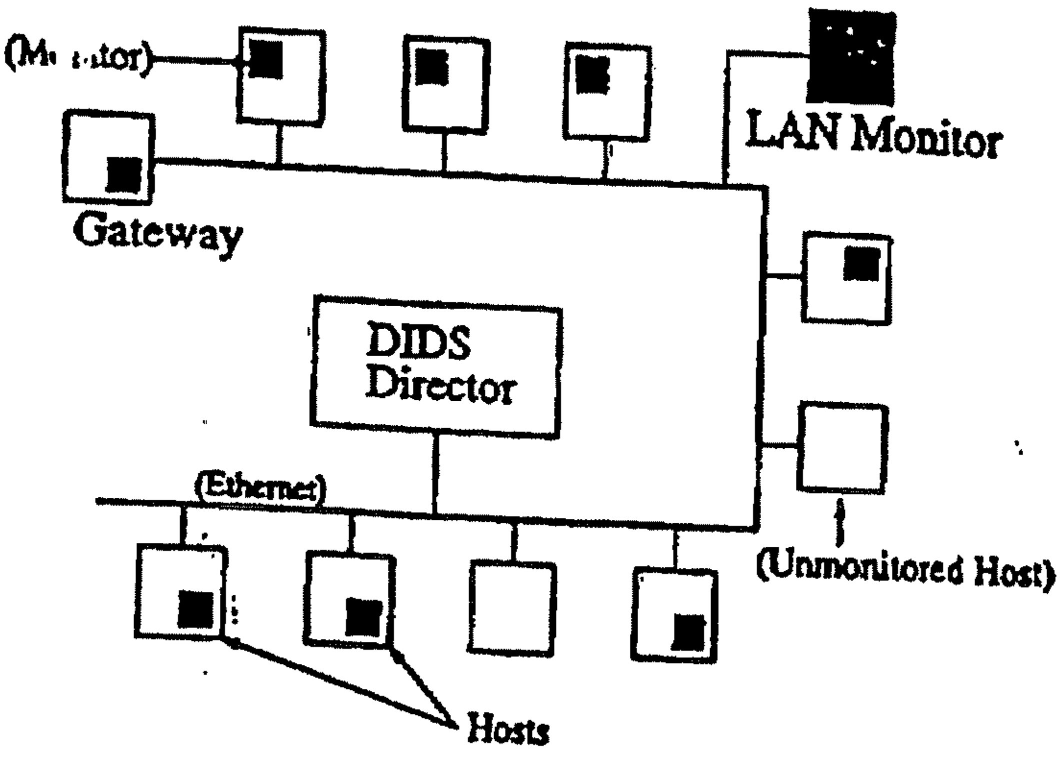


Fig. 1. DIDS Target Environment

Figure 25: (Figure 1 of [DIDS]) showing an overview of DIDS deployed on a computer network.

Figure 1 shows the hosts (square hollow boxes) attached to a network (lines). The black boxes inside the hosts represent the DIDS *host monitors* that ran on each host, examined the host audit trail, and created an event stream of significant happenings or potential intrusions. The large black square at top right of the figure is the LAN Monitor. This was responsible for watching the network looking for intrusions and creating an event stream tracking activities on the network required for correlation. The DIDS LAN Monitor was based on the NSM (described above). Finally, the large rectangle in the middle of Figure 25 is the DIDS Director. This was the centralized component that took events from the LAN Monitors and Host Monitors, drew overall conclusions, and presented them to the user.

The flow of communication is shown more explicitly in Figure 26:

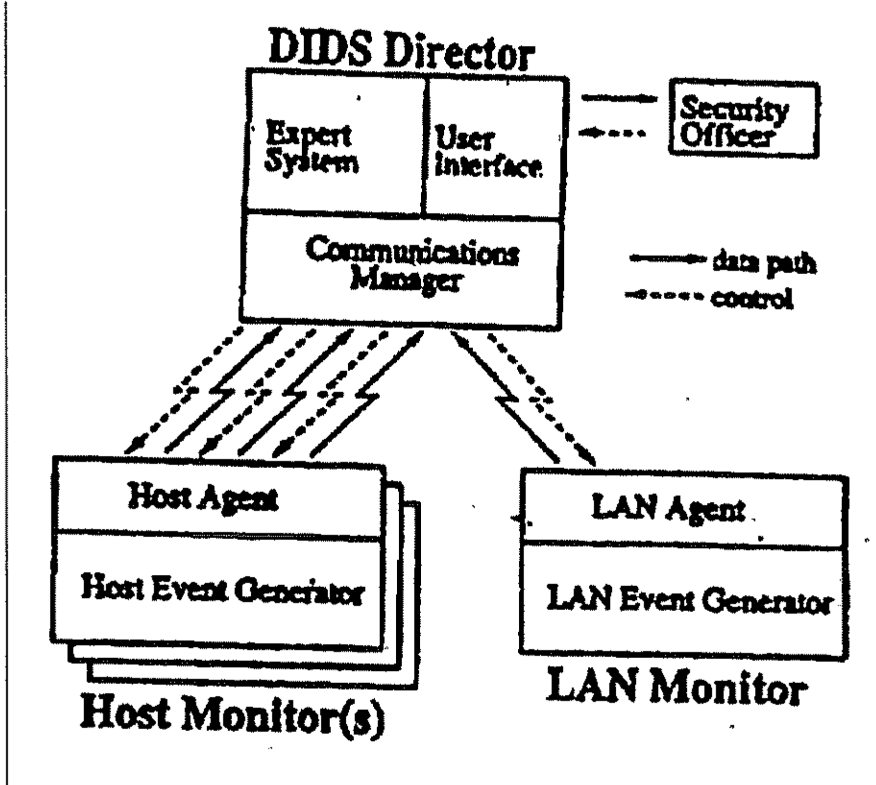


Fig. 2. Communications Architecture

Figure 26: (Figure 2 of [DIDS]) showing communication between the elements of the DIDS system.

The architecture shows the Host Monitor split into a Host Event Generator which reads the system audit trail, applies signatures to it, and creates DIDS event records as needed, and the Host Agent which is responsible for communication with the DIDS director. The LAN Monitor has a similar architecture – a LAN Event Generator, which is basically an NSM, together with a LAN agent which is responsible for interacting with the Director. The Director itself consists of a communication portion which interacts with the various monitors, the Expert System component that did actual correlation and reasoning, and then a user interface which handled communication with the human users of DIDS.

The DIDS Director combined events from the LAN monitors and the Host monitors. It did this based on commonalities amongst the events being reported to it. For example, extensive effort was expended [DRA] to ensure that users were tracked as they moved from host to host so that all their activities could be accounted to a single *Network User ID (NID).*

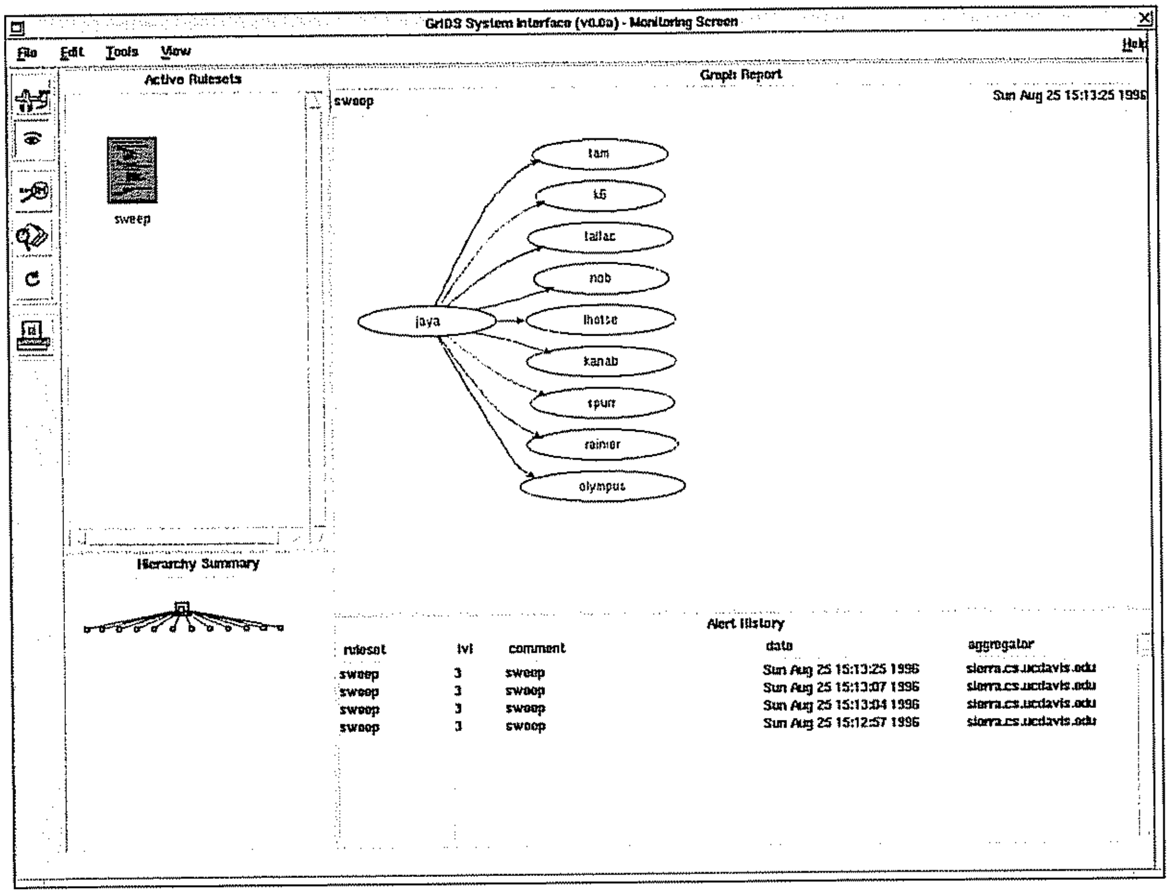
IIg3) Graph-Based Intrusion Detection System (GrIDS)

The GrIDS project at UC Davis began in late 1995, published a first paper in late 1996, by which time the system was essentially implemented, and continued for several more years with diminishing levels of effort. Most descriptions in this report are as of a technical report which described the design of the system in some detail and a version of which was published on the UC Davis website no later than August 27th, 1996. The author of this expert report was also the technical leader of the GrIDS effort.

The GrIDS team worked collaboratively to design the system through 1996. The GrIDS technical report was begun in February of 1996, and largely completed by May of 1996. At some point no later than August 27th 1996, when I gave a presentation about the system to a DARPA PI meeting in Santa Cruz, CA, I placed it on the GrIDS web site (part of the UC Davis computer security laboratory website) where it was available to the world. It was then updated at regular intervals.

The motivation for GrIDS was to push beyond the then state-of-the-art in intrusion detection, and the central direction of that push was to be able to perform intrusion detection across large networks – either the network of a large organization, or perhaps eventually even the entire Internet. The goal was partly to organize, correlate, and display the alerts from individual sensors (network intrusion detection systems or host intrusion detection systems for individual computers), but more importantly, to detect large scale attacks on the network, which might only be detectable by correlating information from multiple locations. The system was especially targeted at scans and worms, but also could put together chains of actions by individual hackers compromising a number of machines in sequence.

GrIDS put together activities into *GrIDS graphs*, an example of which is shown here. I created this picture and presented it at as part of the slideshow presentation at a DARPA Principal Investigators meeting at Santa Cruz in 1996. It is an actual screen shot from a running system in August 1996.



rulesot	lvl	comment	date	aggregator
sweep	3	sweep	Sun Aug 25 15:13:25 1996	sierra.cs.ucdavis.edu
sweep	3	sweep	Sun Aug 25 15:13:07 1996	sierra.cs.ucdavis.edu
sweep	3	sweep	Sun Aug 25 15:13:04 1996	sierra.cs.ucdavis.edu
sweep	3	sweep	Sun Aug 25 15:12:57 1996	sierra.cs.ucdavis.edu

Figure 27: A screenshot of the main monitoring screen in the GrIDS user interface

The large top right panel in the user interface shows a graph of the kind that GrIDS was trying to detect and visualize. A graph is a kind of diagram used by computer scientists to understand a variety of phenomena. Graphs have *nodes* (the labeled ovals in the picture), and *edges* (the lines which join the nodes). In GrIDS, the nodes were always hosts on a network, and the edges represented that some kind of network traffic had been made between those hosts. Typically this would be a TCP connection, but not necessarily. The assertion made by a graph that has been reported to the user interface like this is essentially "this pattern of activity on and between these hosts is all causally related, and it is suspicious". The formation of these graphs was the way GrIDS correlated data from a variety of data sources on the network.

In the particular case shown, the activity is a sweep. The computer named "Jaya" has connected to 9 other computers in a manner that is sufficiently rapid to have caused GrIDS to detect it and label it suspicious.

GrIDS was designed to be a system that could take input from a variety of sources of data. These could be other intrusion detection systems reporting problems on a particular host, filters that translated host logs into GrIDS data, or network systems reporting connections or various kinds of network activity, A special language was defined for expressing all this data in GrIDS graphs. Data that was about a particular host would form a *node report*, while data that was about some kind of network activity between two hosts would form an edge report.

One particular source of data was a network sniffer component that essentially translated packets on the network into GrIDS graph reports that were then consumed by the local graph engine. For example, it would create records when a new network connection was requested, not whether it succeeded or was denied, and so forth. These would become edge reports that went into the lowest level graph engine.

Note the "Hierarchy Summary" panel in the lower left of the screenshot in Figure XXX. An important aspect of GrIDS is that all prior distributed intrusion detection systems (eg DIDS or NIDES) organized processing only on the sensor where the data was initially generated (either from host audit trails or packets), or on a centralized platform that was responsible for all correlation reasoning: reasoning where data from several sensors had to be combined to draw a conclusion. In GrIDS, correlation and inference could be performed across a larger hierarchy of processing modules (called *graph engines* in GrIDS). The idea is illustrated with the following diagram (again taken from the 1996 presentation).

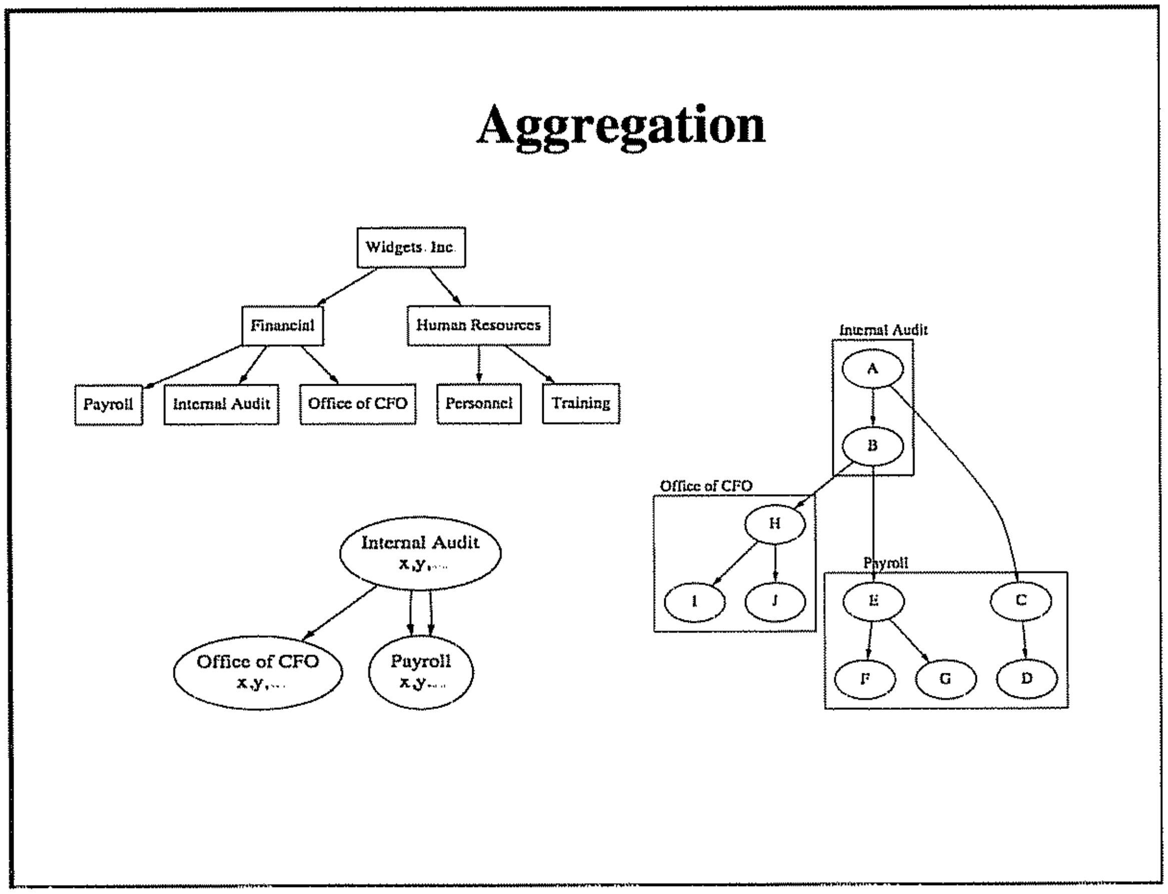


Figure 28: Aggregation of graphs in the GrIDS hierarchy

In the top left, we see part of the organizational chart for a hypothetical enterprise (Widgets, Inc) on which GrIDS might be deployed. The organization is decomposed into a hierarchy of *departments* with sub-departments inside the departments. Thus for example, the "Payroll" department is part of the "Financial" department within Widgets, Inc. These departments are assumed to contain computers (and any computer is in exactly one department). Each department had its own graph engine, which was responsible for doing correlation of activity within that department by forming graphs of suspicious activity. This particular hierarchy has three levels of departments, but GrIDS could have arbitrarily many such levels.

A department might contain a variety of data sources, but only one GrIDS engine to which all the local data-sources would send their reports in the graph language. The graph engine would combine them into graphs.

The graph engine itself was fairly generic. It contained code for maintaining the graph structures, interpreting the graph language, and deciding whether two graphs overlapped (which was a necessary condition for them to get combined). However, the detailed logic for deciding exactly how to build graphs was contained in *rulesets*. GrIDS could run a number of rulesets at the same time, and they would separately build different kinds of graphs according to their own logic. Graphs from one ruleset would never interact or combine with graphs from another ruleset.

The rulesets were written in a rule language that specified under what circumstances nodes and edges would combine into graphs, and graphs combine into larger graphs. Two graphs could only combine if they overlapped in their endpoints (that is they must have at least one host in common). Beyond this, the ruleset could make fairly arbitrary decisions based on the attributes of the nodes and edges in the potentially combining graphs. For an example, it was common for graphs to only combine if the time attribute at nodes where close enough. Thus if two edge reports that might be parts of a possible scan occurred three seconds apart, they could combine. If they occurred two days apart, they couldn't. Indeed graphs timed out after a while, so the older one might not be available to combine.

The ruleset also specified when graphs were suspicious enough to be reported to a user interface. Generally, graphs were decided to be suspicious either if the graph was large (which was typical of worms and scans), or a data-source had labeled one of the nodes or edges as suspicious directly (in which case the entire graph would inherit that suspicion). However, in principle the ruleset allowed the decision about whether a graph was suspicious or not to be made based on arbitrary properties of the graph. Graphs that cut across multiple departments were always passed up to the graph engine of higher departments in the hierarchy, but graphs were also passed up the hierarchy if the ruleset changed a global variable in the graph (which would invariably be the case if the graph had crossed the threshold to be alerted to the user interface, but could also be done at the ruleset's option for other reasons of correlation).

In the lower right of the figure, a worm graph is shown as the worm spreads through the computers in several departments. The worm has started on computer A in the Internal Audit department, and then spread to computer B in the same department, but also to C in the Payroll department. From B it separately spreads to E in Payroll, and also to H in the Office of the CFO. And so forth.

If all the computers infected by the worm were in the same department, GrIDS would be straightforwardly able to build the whole graph, notice that it was large, and decide it was suspicious and worm-like. However, because the graph is split between multiple bottom level departments, each of them cannot see the whole picture. Indeed the graph engine for the Payroll department cannot even tell that the CD graph and the EFG graph are related rather than separate, since that connection occurred in the Internal Audit department.

The way GrIDS solves this problem is that all graphs that cross a department boundary are passed up to the next higher department in the hierarchy (in this case Financial). However, to make the system more scalable, GrIDS did not pass up the entire graph (which could be very large in a realistic system), but rather just a reduced graph in which the department appeared as a single node. Special attributes of this reduced graph carried information like the number of nodes (hosts) and edges in the graph. This allowed the Financial department to build the graph shown in the lower left of Figure 28. However, crucially, the attributes of all the graphs could be added together and thus the Financial department graph engine could tell that the graph was in fact large enough to cross the threshold and be considered suspicious.

This last figure displays an actual three level hierarchy running on notional departments within the computer security lab at UC Davis in 1996. This again is from a slide in the presentation to the Santa Cruz PI meeting [Santa Cruz PI].

All of the above was described in detail in the May 1997 Technical Report available on the web. [GrIDS].

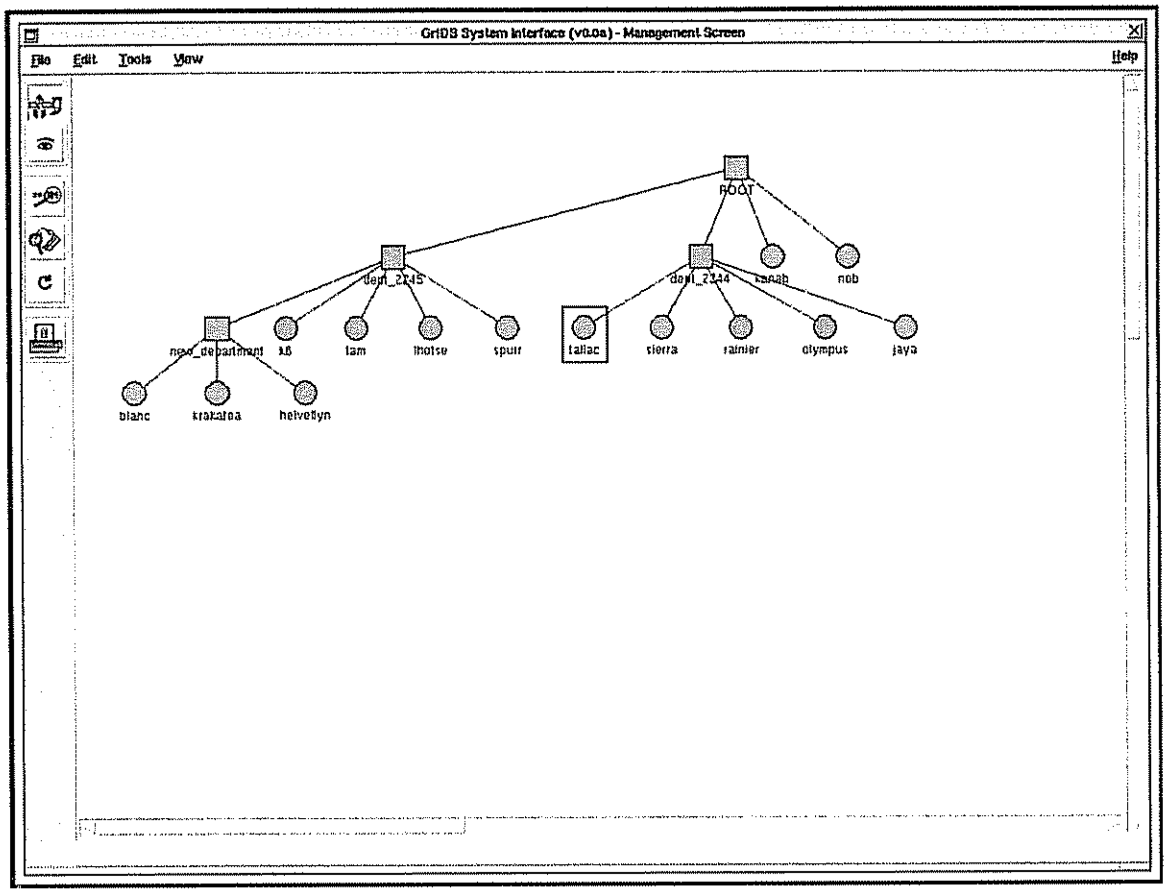


Figure 29: A screenshot of a running GrIDS user interface showing the hierarchy available for management.

IIi) DARPA PI meetings and CIDF

It is important to understand that from 1996 on, the Defense Advanced Research Projects Agency, under the auspices of Program Manager Teresa Lunt, began a substantial ramp up of funding into intrusion detection research. This led to the formation of a community of researchers who regularly attended DARPA PI meeting. The researchers working on Ms Lunt's program who regularly attended PI meetings included at a minimum Karl Levitt and myself from UC Davis (GrIDS), Phil Porras and Al Valdes from SRI (EMERALD), Felix Wu of NCSU and Frank Jou of MCNC (JiNao), Dan Schnackenberg of Boeing (IDIP), Brian Tung of ISI (CRISIS), Maureen Stillman of Oddyssey Research, Rich Feiertag of Trusted Information Systems, and Cliff Kahn of the Open Group, as well as numerous others. We were required to disclose our technical progress to one another at PI meetings (including in Santa Cruz, CA in August 1996, and in Savannah, GA in February 1997)

Additionally, Teresa Lunt, our Program Manager required our projects to work together to develop the Common Intrusion Detection Framework (CIDF), an effort to develop common APIs and means for intrusion detection systems to interoperate with each other. A short history of this effort is available from [CIDF]

> "The goal of the Common Intrusion Detection Framework is a set of specifications which allow
>
> - "different intrusion detection systems to inter-operate and share information as richly as possible,
> - "components of intrusion detection systems to be easily re-used in contexts different from those they were designed for.
>
> "The CIDF working group came together originally in January 1997 at the behest of Teresa Lunt at DARPA in order to develop standards to accomplish the goals outlined in the previous section. She was particularly concerned that the various intrusion detection efforts she was funding be usable and reusable together and have lasting value to customers of intrusion detection systems.
>
> "During the life of the effort, it became clear that this was of wider value than just to DARPA contractors, and the group was broadened to include representatives from a number of government, commercial, and academic organizations. After the first few months, membership in the CIDF working group was open to any individuals or organizations that wished to contribute. No cost was involved (except to defray meeting expenses).
>
> "Major decisions were made at regular (every few months) meetings of the working group. Those decisions were made by rough consensus of all attendees. That is, the meeting facilitator attempted to reach consensus, but in situations where only one or two individuals were protesting a decision, they were overruled in the interest of efficiency. No decisions were taken in the face of opposition from a sizeable minority, rather the issue was tabled for further consideration. Meetings were fun and the working group had a good time doing this (well, most of them, anyway).
>
> "In between meetings, most of the writing was done by small subgroups or individuals. Their text was brought back for approval/changes at meetings. Discussions were also carried on in the working group mailing list, but few decisions were made that way.

An overview of the architecture of the resulting system was also in that document:

All CIDF components deal in *gidos* (generalized intrusion detection objects) which are represented via a standard common format. Gidos are data that is moved around in the intrusion detection system. Gidos can represent events that occurred in the system, analysis of those events, prescriptions to be carried out, or queries about events.

CIDF defines four interfaces that CIDF components may implement:

	Push-style	Pull-style
Producer	Produces gidos when it wants to, typically in response to events.	Produces gidos when queried.
Consumer	Mates with push-style producer.	Mates with pull-style producer.

Each of these interfaces takes two forms: a callable form, which permits reuse of the component, and a protocol form, which permits the component to interoperate with other CIDF components.

CIDF defines several types of preferred components:

* Event generators
* Analyzers
* Databases
* Response units

Figure 1.1 presents a schematic view of these components in a hypothetical intrusion detection system. The solid boxes labeled E1, E2, A1, A2, D, etc represent the various components of some hypothetical intrusion detection system. It is convenient to think of these as objects in the object-oriented programming sense (this does not dictate an implementation in an object-oriented language or framework).

Phillip Porras and I were two of the coauthors of that document and the CIDF group had been working together on that architecture and those interfaces for ten months before the statutory bar date, and 20 months before he filed the application that became the 338 patent.

In my opinion, the CIDF standard as it emerged had large overlaps with the patent claims filed by SRI. At no time did Mr Porras reveal to me that he was filing patents covering broad areas of our joint work on CIDF.

IIj) Emerald

In 1996 SRI resumed intrusion detection work under new leadership. With Teresa Lunt seconded to DARPA, Phillip Porras came on board to head the intrusion detection group, working with veterans of IDES and NIDES including Peter Neumann and Al Valdes. It was during the Emerald project that the patents in suit were to be filed.

VII) Reservations of Rights

I reserve the right to amend or supplement the statement based on further discovery and preparation in this action, including my review of any expert statement submitted on behalf of SRI, and review of material currently designated confidential.

Dated: April 21, 2006

[illegible]

Stuart Staniford